les suites

Introduction/Motivation

15 octobre 2018

table des matières

Suites

Définition

On appelle suite numérique une application de $\mathbb{N}$ dans $\mathbb{R}$.

Notation

Plutôt que d'utiliser une notation fonctionnelle, on utilise souvent la notation indicée. Si u est une suite, on note u_n le terme numéro n de la suite u.

exemple

$$u_n = 3^n - 2^n$$

$$v_n = 1 + 2 + 3 + ... + n$$

Il est possible de définir une suite à l'aide d'une formule donnant directement le terme d'indice n en fonction de cet indice n.

exemple

$$u_n = 3^n - 2^n$$

Il est possible de définir une suite par *récurrence* le terme d'indice n etant calculé en fonction d'un ou plusieurs termes précédents.

Il est possible de définir une suite par *récurrence* le terme d'indice n etant calculé en fonction d'un ou plusieurs termes précédents.
Bien sûr, dans ce cas, il faut préciser la valeur du ou des premier(s) terme(s).

Il est possible de définir une suite par *récurrence* le terme d'indice n etant calculé en fonction d'un ou plusieurs termes précédents.
Bien sûr, dans ce cas, il faut préciser la valeur du ou des premier(s) terme(s).

exemples

La célébrissime suite de Héron :

$$u_0 = 2$$

$$\forall n \in \mathbb{N} \quad u_{n+1} = \frac{(\frac{2}{u_n} + u_n)}{2}$$

Dans ce cas le calcul du terme d'indice n nécessite le calcul du termes précédents.

Il est possible de définir une suite par *récurrence* le terme d'indice n etant calculé en fonction d'un ou plusieurs termes précédents.
Bien sûr, dans ce cas, il faut préciser la valeur du ou des premier(s) terme(s).

exemples

La célébrissime suite de Héron :

$$u_0 = 2$$

$$\forall n \in \mathbb{N} \quad u_{n+1} = \frac{(\frac{2}{u_n} + u_n)}{2}$$

Dans ce cas le calcul du terme d'indice n nécessite le calcul du termes précédents. la programmation du calcul du terme d'indice n, s'écrit assez bien de manière *récursive*

Héron maladroit


```
def heron_maladroit(n):
    """
    compute recursively the term of
    index n of the Heron sequence

    :param n: the index of the term
    :type n: int
    :rtype: float
    :UC: n is a non negative int. if not raise AssertError
    :Examples:

    >>> heron_maladroit(0)
    2.0
    >>> heron_maladroit(1)
    1.5
    >>> heron_maladroit(5)
    1.414213562373095
    """
    assert type(n)==int
    assert n>=0
    if n==0:
        return 2.0
    else:
        return ((2.0/heron_maladroit(n-1))+heron_maladroit(n-1))/2.
```

des suites pour compter

L'implantation de la fonction `heron_maladroit` va permettre d'illustrer une utilisation qu'on peut faire des suites en informatique.

compter le nombre d'appel

On souhaite compter le nombre d'appels à la fonction `heron_maladroit` effectués lors du calcul du terme d'indice n

On définit la suite a par a_n désigne le nombre d'appels à la fonction `heron_maladroit` effectués lors du calcul de `heron_maladroit(n)`

un fait évident

$$a_0 = 1$$

une autre évidence

$$a_n = 1 + a_{n-1} + a_{n-1} = 2 * a_{n-1} + 1$$

Et voilà !
On a défini une suite par récurrence.
Biensur, il serait intéressant et utile d'avoir une expression de a_n en fonction de n.
Ce n'est

- ni toujours facile
- ni toujours possible !

On verra un peu plus loin comment on traite ce genre de suite. ici, on se contente de deviner le résultat et de le démontrer par récurrence. Voici donc l'hypothèse de récurrence :

$$\mathcal{H}_n : \qquad a_n = 2^{n+1} - 1.$$

on vérifie aisément que

- $\mathcal{H}_0$ est vraie
- si $\mathcal{H}_n$ est vraie alors $\mathcal{H}_{n+1}$ est aussi vraie

Donc le calcul de `heron_maladroit(24)` nécessite $2^{25} - 1 = 33554431$ appels à `heron_maladroit` ce qui est beaucoup trop !

L'utilisation des suites va permettre de calculer les coûts de nos programmes afin de les rendre performants et "verts".
Le problème dans `heron_maladroit` provient du fait qu'il y a deux appels récursifs pour calculer la même valeur.
En transformant un peu le programme, on va améliorer les performances...

Héron amélioré


```
def heron_ameliore(n):
    """
    compute recursively the term of
    index n of the Heron sequence

    :param n: the index of the term
    :type n: int
    :rtype: float
    :UC: n is a non negative int. if not raise AssertError
    :Examples:

    >>> heron_ameliore(0)
    2.0
    >>> heron_ameliore(1)
    1.5
    >>> heron_ameliore(5)
    1.414213562373095
    """
    assert type(n)==int
    assert n>=0
    if n==0:
        return 2.0
    else:
        h_prec=heron_ameliore(n-1)
        return ((2.0/h_prec)+h_prec)/2.
```

On définit la suite b par b_n désigne le nombre d'appels à la fonction `heron_ameliore` effectués lors du calcul de `heron_ameliore(n)`.

un fait évident

$b_0 = 1$

une autre évidence

$$b_n = 1 + b_{n-1}$$

La vérification que

$$\forall n \in \mathbb{N} \qquad b_n = n + 1$$

est laissée en exercice...

L'étude des suites va permettre d'expliquer certains phénomènes qui pourraient sembler étranges à première vue. Voici un exemple frappant :

$$u_0 = \frac{1}{3}$$

$$\forall n \in \mathbb{N} \qquad u_{n+1} = 4 * u_n - 1$$

programmons en python cette suite

```
def instable(n):
    """
    compute recursively the term of
    index n of the sequence u_{n+1}=4 u_n -1
    with u_0=1/3

    :param n: the index of the term
    :type n: int
    :rtype: float
    :UC: n is a non negative int.
      if not raise AssertError
    :Examples:

    >>> instable(0)
    0.3333333333333333
    >>> instable(1)
    0.33333333333333326
    >>> instable(10)
    0.3333333333139308
    """
    if n==0:
        return 1/3
    else:
        return 4*instable(n-1) -1
```

Or on peut démontrer par récurrence que cette suite est constante ...
Comment expliquer ceci ?

prouvons par récurrence que si $u_0 = \frac{1}{3} + \epsilon$ alors $u_n = \frac{1}{3} + 4^n\epsilon$

ce résultat, conjugué à notre connaissance de IEEE 754 explique le phénomène.

Résolution du problème des tours de Hanoï

Entrée :

- $n \in \mathbb{N}$ le nombre de disques à déplacer,
- d le tas où ils se trouvent,
- a le tas où on doit les mettre.

Action

les déplacements des disques pour les amener sur le tas d'arrivée

```
si n>0
  aux := le piquet différent de d et a
  hanoi(n-1,d,aux)
  deplacer_disque(d,a)
  hanoi(n-1,aux,a)
fin si
```

Idée décompte du nombre de déplacements ?

Le décompte des déplacements

On appelle d la suite définie par : pour tout entier n le terme d_n désigne le nombre de déplacements nécessaires pour déplacer n disques.

un fait

s'il n'y a pas de disques à déplacer, il faut faire 0 déplacement. donc $d_0 = 0$

un autre fait

si n est un entier naturel non nul :

$$d_n = d_{n-1} + 1 + d_{n-1} = 2d_{n-1} + 1$$

c'est la même relation de récurrence que pour la suite a, toutefois, le terme initial est différent. On peut démontrer par récurrence que $d_n = 2^n - 1$

Nombre de fichiers de n octets qui sont des chaînes UTF-8 valides ?

Nbre octets codants	Format du code
1	`0xxxxxxx`
2	`110xxxxx 10xxxxxx`
3	`1110xxxx 10xxxxxx 10xxxxxx`
4	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

Table – Format du codage UTF-8

la relation de récurrence

on suppose $n \geq 4$

le premier caractére du fichier utf-8 est codé

- soit sur un octet, dans ce cas on a 2^7 choix possibles pour cet octet. D'autre part, le reste forme un fichier utf-8 valide de $n-1$ octets. Il y a u_{n-1} tels fichiers.
- soit sur deux octets, dans ce cas on a 2^{11} choix possibles pour ces octets. D'autre part, le reste forme un fichier utf-8 valide de $n-2$ octets. Il y a u_{n-2} tels fichiers.
- soit sur trois octets, dans ce cas on a 2^{16} choix possibles pour ces octets. D'autre part, le reste forme un fichier utf-8 valide de $n-3$ octets. Il y a u_{n-3} tels fichiers.
- soit sur quatre octets, dans ce cas on a 2^{21} choix possibles pour ces octets. D'autre part, le reste forme un fichier utf-8 valide de $n-4$ octets. Il y a u_{n-4} tels fichiers.

$$\forall n \in \mathbb{N} \qquad n \geq 4 \Longrightarrow u_n = 2^7 u_{n-1} + 2^{11} u_{n-2} + 2^{16} u_{n-3} + 2^{21} u_{n-4}$$

les premiers termes

- il y a un seul fichier utf-8 valide de zéro octets.
- il y a 2^7 fichiers utf-8 valides représentant une chaîne d'un caractère.
- - il y a $2^7 \times 2^7$ fichiers utf-8 valides de deux octets représentant une chaîne de longueur 2
  - il y a 2^{11} fichiers utf-8 valides de deux octets représentant une chaîne de longueur 1
- - il y a $2^7 \times 2^7 \times 2^7$ fichiers utf-8 valides de trois octets représentant une chaîne de longueur 3
  - il y a $2(2^7 \times 2^{11})$ fichiers utf-8 valides de trois octets représentant une chaîne de 2 caractères,
  - il y a 2^{16} fichiers utf-8 valides de trois octets de longueur représentant une chaîne de 1 caractère.

$$u_0 = 1$$

$$u_1 = 128$$

$$u_2 = 16384 + 2048 = 18432$$

$$u_3 = 2097152 + 524288 + 65536 = 2686976$$

```
def number_of_utf8_files_length(n):
    """ number_of_files_utf8_length
    compute the term of
    index n of the number of files of length n
    which are utf-8 valid

    :param n: the index of the term
    :type n: int
    :rtype: int
    :UC: n is a non negative int. if not raise AssertError
    :Examples:

    >>> number_of_utf8_files_length(1)
    128
    >>> number_of_utf8_files_length(0)
    1
    >>> number_of_utf8_files_length(5)
    57176752128
    """
    assert type(n)==int
    assert n>=0
    vect=list(reversed([128,2048,65536,2097152]))
    u=[1,128,18432,2686976]
    while n>0:
        u.append(sum( [vect[i]*u[i] for i in range(4)]))
        u.pop(0)
        n=n-1
    return u[0]
```